Tomasulo's algorithm

This scheme, first found on the IBM 360/91, was invented by Robert Tomasulo, and it tracks when operands for instruction are available to minimise RAW hazards. Furthermore, Tomasulo's algorithm introduces register renaming in hardware to minimise WAW and WAR hazards.

> **Register renaming**
>
> Register renaming is a technique that aims at eliminating name dependences by renaming all destination registers, including those with a pending read or write for an earlier instruction, so that the out-of-order write does not affect any instructions that depend on an earlier value of an operand.

In Tomasulo's scheme, register renaming is provided by *reservation stations*, which buffer the operands of instructions waiting to issue. The basic idea is that a reservation station fetches and buffers an operand as soon as it is available, eliminating the need to get the operand from a register.

In addition, pending instructions designate the reservation station that will provide their input.

Finally, when successive writes to a register overlap in execution, only the last one is actually used to update the registe.

Since there can be more reservation stations than real registers, the technique can even eliminate hazards arising from name dependences that could not be eliminated by a compiler.

The use of reservation stations leads to two important properties:

1. Hazard detection and execution control are distributed: the information held in the reservation stations at each functional unit determines when an instruction can begin execution at that unit.
2. Results are passed directly to functional units from the reservation stations where they are buffered, rather than going through the registers. This bypassing is done with a common result bus[1] that allows all units waiting for an operand to be loaded simultaneously.

Instruction processing

There are only three steps that every instruction needs to go through in Tomasulo's algorithm:

Step	Description
Issue	Get the next instruction from the head of the instruction queue. If there is a matching reservation station that is empty, issue the instruction to the station with the operand values, if they are currently in the registers. If there is not an empty reservation station, then there is a structural hazard and the instruction stalls until a station or buffer is freed. If the operands are not in the registers, keep track of the functional units that will produce the operands. This steps renames registers, eliminating WAR and WAW hazards.[2]
Execute	If one or more of the operands is not yet available, monitor the common data bus while waiting for it to be computed. When an operand becomes available, it is placed into any reservation station awaiting it. When all operands are available, the operation can be executed at the corresponding functional unit. By delaying the instruction execution until the operands are available, RAW hazards are avoided[3] [4]. Loads and stores require a two-step execution process: the first step computes the effective address when the base register is available; the effective address is then places in the load or store buffer. Loads in the load buffer execute as soon as the memory unit is available. Stores in the store buffer wait for the value to be stored before being sent to the memory unit. Loads and stores are maintained in program order through the effective address calculation, which will help to prevent hazards through memory. egoTo preserve exception behaviour, no instruction is allowed to initiate execution until all branches that precede the instruction in program order have completed.
Write result	When the result is available, write it on the CDB and from there into the registers and into any reservation stations—including store buffers—waiting for this result. Stores are buffered in the store buffer until both the value to be stored and the store address are available, then the result is written as soon as the memory unit is free.

The data structures that detect and eliminate hazards are attached to the reservation stations, to the register file and to the load and store buffers with slightly different information attached to different objects. These tags are essentially names for an extended set of virtual registers used for renaming. Once an instruction has issued and is waiting for a source operand, it refers to the operand by the reservation station number where the instruction what will write the register has been assigned. Because there are more reservation stations than actual register numbers, WAW and WAR hazards are eliminated by renaming results using reservation station numbers.

Uses of the algorithm

Tomasulo's scheme was unused for many years after its introduction with the IBM 360/91, but was widely adopted in multiple-issue processors starting in the 1990s

for several reasons:

1. Although Tomasulo's algorithm was designed before caches, the presence of caches—with their inherently unpredictable delays—has become one of the major motivations for dynamic scheduling. Out of order execution allows the processors to continue executing instructions while awaiting the completion of a cache miss, thus hiding all or part of the cache miss penalty.
2. As processors became more aggressive in their issue capability and designers more concerned with the performance of difficult-to-schedule code, techniques such as register renaming, dynamic scheduling and speculation became more important.
3. It can achieve high performance without requiring the compiler to target code to a specific pipeline structure, a valuable property in the era of shrink-wrapped mass market software.

---

1. On the IBM 360/91, this is called the *Common Data Bus*, or CDB. Sometimes we will be referring to a generic common result bus with this name as well. ↩
2. This step is sometimes called *dispatch* in a dynamically scheduled processor. ↩
3. Some dynamically scheduled processors call this step *issue*. ↩
4. Several instructions could become ready in the same clock cycle for the same functional unit. If more than one instruction is ready for a single functional unit, the unit will have to choose among them. ↩